A High Resolution Finite Volume Method for Efficient Parallel Simulation of Casting Processes on Unstructured Meshes *

Douglas B. Kothe † Robert C. Ferrell ‡ John A. Turner § S. Jay Mosso ¶

Abstract

We discuss selected aspects of a new parallel three-dimensional (3-D) computational tool for the unstructured mesh simulation of Los Alamos National Laboratory (LANL) casting processes. This tool, known as **Telluride**, draws upon robust, high resolution finite volume solutions of metal alloy mass, momentum, and enthalpy conservation equations to model the filling, cooling, and solidification of LANL castings. We briefly describe the current **Telluride** physical models and solution methods, then detail our parallelization strategy as implemented with Fortran 90 (F90). This strategy has yielded straightforward and efficient parallelization on distributed *and* shared memory architectures, aided in large part by new parallel libraries **JTpack90** [21] for Krylov-subspace iterative solution methods and **PGSLib** [7] for efficient gather/scatter operations. We illustrate our methodology and current capabilities with source code examples and parallel efficiency results for a LANL casting simulation.

1 Introduction

We are currently pursuing the development of a comprehensive and robust casting simulation tool, known as **Telluride** [12], which is being designed to model the molten fluid flow, heat flow, solidification, species transport, and interface dynamics present in metal alloy casting processes at LANL foundries. To be value-added, **Telluride** must not only integrate all these relevant physical processes, it must also incorporate the latest advances in numerical algorithms and solidification theory. In addition, the computational resources commanded by casting process simulation necessitate efficient parallel execution on current high performance computing architectures.

Driven by increasing demands on quality and control of microstructure, solidification theory and modeling provide the basis for influencing microstructure and improving the quality of cast products. For example, a common occurrence in castings is the local variation of microstructure, which can result in compositional and property variation throughout the entire part. Such defects are difficult to eliminate once they are cast into the part, tending to persist even after final forming. We anticipate that **Telluride** will have the potential to improve casting practices, reduce foundry costs, and provide a means to advance the theory and understanding of alloy solidification.

*Supported by the Department of Energy Accelerated Strategic Computing Initiative Program.
†LANL, Fluid Dynamics Group T-3, MS B216, Los Alamos, NM 87545 (*dbk@lanl.gov*).
‡Cambridge Power Computing Associates, Ltd., 2 Still St., Brookline, MA 02146 (*ferrell@cpca.com*).
§LANL, Transport Methods Group X-TM, MS B226, Los Alamos, NM 87545 (*turner@lanl.gov*).
¶LANL, Hydrodynamic Applications Group X-HM, MS F663, Los Alamos, NM 87545 (*sjm@lanl.gov*).

2 Physical Model and Solution Method

A realistic model for metal alloy casting processes requires descriptions for many physical phenomena: incompressible free surface flow of the molten metal during the fill process, interfacial surface tension at the molten metal free surface, solidification and melting phase change rates of multiple species alloys possessing an arbitrary phase diagram, alloy species liquid and solid phase transport, and microscopic mushy zone effects, to name a few. We follow the methodology of Beckermann [3], in which alloy species mass, momentum, and energy equations are volume-averaged in a traditional multiphase approach.

Metal alloy mass, momentum, and energy transport is modeled with a simplified version of the volume-averaged two-phase model of Beckermann [17, 3]. In formulating the model equations, we currently assume that the solid phase is stationary, the solid and liquid phases are in thermal equilibrium, liquid species concentrations are equal to their interfacial averages, and finite-rate macroscopic species diffusion is negligible. See [18] for details of the current **Telluride** alloy solidification models.

Our incompressible flow algorithm builds upon our past work on two-dimensional free surface flows [11] having interfacial surface tension [5]. We have increased the algorithm accuracy and robustness by incorporating the advances of Bell and coworkers [4] in devising high resolution projection method solutions of the Navier-Stokes equations coupled with modern interface tracking algorithms. This approach has yielded high-fidelity flow solutions that are fully second-order in time and space [19].

We have extended projection-based Navier-Stokes solution methods to 3-D unstructured grids without unnecessarily sacrificing robustness, accuracy, or efficiency. Our current approach has borrowed from the innovative techniques of Barth [1], an example being least-squares reconstruction schemes. We have also extended a 3-D unsplit advection technique [20] to unstructured meshes, which has allowed consistent use of high-order monotone advection in incompressible flows.

Finally, we have extended our volume tracking algorithms to 3-D generalized hexahedral grids [13]. Interfaces are tracked on generalized hexahedral meshes and localized over a one cell width for each time step. Interfaces are assumed to be locally planar within each cell, giving a globally piecewise planar approximation to the actual interface topology.

3 Software Design Issues

We now discuss our software design philosophies and goals and our implementation using object-based F90 [6]. We also discuss briefly our coordination of a development team tasked to engineer efficient software within stringent project constraints [16].

3.1 Design Philosophy

Many important decisions confronted while engineering the **Telluride** software have been guided by our principal design goals of seamless portability, functionally-based modularity, and efficient parallelism. Since current architectures change on a yearly basis, software longevity will not be realized if design and implementation is targeted toward efficient execution on a specific architecture. The **Telluride** software has therefore been implemented in strict adherence to a language standard, chosen to be F90. By commiting to languages that have formal standards, software portability can be realized if compiler availability is widespread and reliability is high. To date, our commitment to F90 as the principal programming language has resulted in successful simulations of casting processes on a long and varied list of computing platforms.

3.2 Implementation via Object-Based Fortran 90

Programming languages are generally considered to be object-oriented (OO) if constructs are provided to support data abstraction, information hiding, inheritance, and templates [14]. F90 explicitly provides for the expression of data abstraction and information hiding, and indirectly allows for some aspects of inheritance and templates. In this regard, F90 might more appropriately be considered an object-based (OB) or functionally object-oriented (F/OO) language [14, 22]. We are currently finding useful many new features offered by F90: free-form source, concise array syntax, portable constructs for precision (kind numbers), data abstraction with derived types, modules and their associated information hiding, argument checking via module procedures and interface blocks, polymorphism via generic procedures, pointered and allocatable variables, and a rich variety of powerful intrinsics. By remaining active in the Fortran programming community, we are confident we will impact the changes, improvements, and additions that will (and should) occur as F90 evolves toward F95 and F2K.

3.3 Team Software Development Practices

Each team member is responsible for one or more modules, defined as a procedure or set of procedures that performs some specific task. Each module has a static and well-defined purpose and interface. This approach allows parallel and independent module development that is not obtrusive to other modules, and is standard practice in many successful commercial software endeavors [15]. Our modules tend to be arranged according to their functionality (e.g., a phase change module, a fluid flow module, etc.), not their data (as in many OO projects), hence the overall design strategy is F/OO.

The **Telluride** modules are constructed with one or more F90 modules, each containing one or more module procedures. The F90 modules are defaulted private, i.e., only the input and output are accessible (public) to the outside world (calling procedure). By containing procedures within modules, they can be hidden, their calling arguments can be optional and/or checked by the compiler, and polymorphism (via generic procedures) can be exploited. By using well-defined interfaces, data structures within modules can change without prior approval from the calling procedure.

Daily functions of the software development team responsible for the design and implementation of **Telluride** and related modules (**JTpack90**, **PGSLib**) are coordinated according to published proven practices [15]. Our software (currently numbering ~50K lines of source code) is maintained with the concurrent versions system (CVS)[1]. We do *not* have a principal "code librarian", i.e., all team members are encouraged to commit modifications to the central source code repository on a regular basis. CVS enables easy extraction of prior versions, and maintains an "audit trial" of the software evolution.

3.4 Example: Mesh Connectivity and Cell Geometry Data Structures

We define parameters for *kind numbers* (essential for portability) and mesh attributes,

```
! ndim - physical dimensions; nfc - faces per cell; nvc - vertices per cell
integer,          parameter :: int_kind = KIND(1), real_kind = KIND(1.0d0)
integer(int_kind), parameter :: ndim = 3, nfc = 6, nvc = 8
```

which enable each **Telluride** cell to be considered a *logical cube*. By allowing cell face vertices to coincide in physical space, this logical cube definition supports all relevant 3-D

[1] See `www.loria.fr/~molli/cvs-index.html` for further information on CVS.

cell types (hex, tet, prism, or pyramid) without cell-specific source code. Given the above parameters, a MESH_CONNECTIVITY derived type is defined for each cell:

```
type MESH_CONNECTIVITY
   integer(int_kind), dimension(nfc) :: Ngbr_Cell, Ngbr_Face
   integer(int_kind), dimension(nvc) :: Ngbr_Vrtx
   integer(int_kind)                 :: Ngbr_PE_Flag
end type MESH_CONNECTIVITY
```

Here, for example, components Ngbr_Cell(f) and Ngbr_Face(f) store the cell and face numbers, respectively, across face f of the reference cell. We also define, for each cell, a CELL_GEOMETRY derived type,

```
type CELL_GEOMETRY
   real(real_kind), dimension(ndim,nfc) :: Face_Normal, Face_Centroid
   real(real_kind), dimension(nfc)      :: Face_Area, Halfwidth
   real(real_kind), dimension(ndim)     :: Centroid
   real(real_kind)                      :: Volume
end type CELL_GEOMETRY
```

which stores all physical cell geometry information. Arrays of these derived types are then declared, which are pointered so their size (ncells) can be determined and allocated dynamically at execution time. Once allocated, array syntax is used for conciseness and readability, e.g., Cell%Volume represents the cell volume array. One drawback of this data structure is that adjacent cell face information is stored redundantly. Many of our data structure choices have placed more importance of conciseness, minimal indirect addressing, and efficient parallelism rather than minimal memory usage.

4 Parallelization Strategy

Our parallelization strategy is quite simple: explicitly decompose and distribute the global **Telluride** mesh across all processors available to perform work on the problem at hand. This strategy is independent of the processor's direct memory access capabilities: local (distributed memory systems) or global (shared memory systems). We have therefore chosen to explicitly program for parallelism, rather than relying upon parallelism via compiler directives (as in High Performance Fortran[2]) or parallelism switches. Explicit parallelism demands greater initial software design and development, but results in more portable and efficiently parallelized software.

We have designed parallelism into our software by separating all communication from computation, then parallelizing the communication via the explicit passing of messages between processors. Message passing, accomplished by calls to the MPI library [8], is necessary when the requested data does not reside in local memory owned by the current processor. For the unstructured meshes utilized by **Telluride**, indirect addressing is required to retrieve neighboring cell information. For example, the following code

```
FACE_LOOP: do f = 1,nfc
   CELL_LOOP: do i = 1,ncells
      Neighbor_Volume(f,i) = Cell(Mesh(i)%Ngbr_Cell(f))%Volume
   end do CELL_LOOP
end do FACE_LOOP
```

[2]See www.crpc.rice.edu/HPFF/home.html for further information on HPF.

returns in array Neighbor_Volume the volume of cell (face) neighbors. This information will not be available to the processor owning cell i if the data for the face neighbor f of cell i resides on another processor. The needed data must first be retrieved from all relevant processors into a local buffer.

Explicit parallelism of this gather operation is accomplished as follows: buffers to hold the incoming and outgoing off-processor data are first allocated; outgoing buffer data is then assimilated and sent; off-processor data is received into the incoming buffer; and, finally, the data is gathered from either the incoming buffer or the original source array. Rather than inserting these constructs wherever indirect addressing operations are needed, we have replaced them with calls to various gather/scatter module procedures. For example, the loop above now becomes:

```
use gs_module, only: EE_GATHER
call EE_GATHER (Neighbor_Volume, Cell%Volume, Mesh)
```

where EE_GATHER is a generic module procedure (in gs_module) that performs all the necessary indirect addressing and message passing functions required to gather Cell%Volume data and return it in Neighbor_Volume.

By invoking gather/scatter module procedures, platform-specific explicit parallelism (message passing) is effectively hidden, instead of being littered throughout the entire source. Communication is also decoupled from all loops performing real computation, which allows compiler optimization to efficiently fuse large code blocks. The principal drawback to this approach is the local allocation of temporary "container arrays" required to hold the output returned by the gather/scatter procedures. We have traded memory in return for modular, portable, and efficient parallelization and computation loops that can be highly optimized.

4.1 Gather/Scatter Modules

To illustrate the functionality of our gather/scatter modules, consider the example source code below, taken from our current gather module. First, we define an EE_GATHER generic procedure that allows the host application to gather scalar or vector data that is of type integer, logical and single/double precision real. This polymorphism allows the applications programmer to use only the EE_GATHER calling protocol, regardless of the data being gathered. Consider the GATHER_DOUBLE module procedure below, which gathers double precision real scalar data from array Src into array Dest:

```
SUBROUTINE GATHER_DOUBLE (Dest, Src, Mesh)
  implicit none
  real(double_kind),       dimension(:,:),          intent(OUT) :: Dest
  type(MESH_CONNECTIVITY), dimension(SIZE(Dest,2)), intent(IN) :: Mesh
  real(double_kind),       dimension(:),            intent(IN) :: Src
  integer(int_kind) :: c, f
  FACE_LOOP: do f = 1,SIZE(Dest,1)
     Dest(f,:) = Src(Mesh%Ngbr_cell(f))
  end do FACE_LOOP
  return
END SUBROUTINE GATHER_DOUBLE
```

This simple procedure is merely a wrapper around the indirect addressing code shown in the NEIGHBOR_VOLUME loop above. If the memory is distributed across processors, however,

explicit parallelization of this procedure is *not* trivial. Parallel versions of our gather/scatter procedures rely upon **PGSLib** to do the interprocessor communication, as shown next.

4.2 Parallel Gather/Scatter with PGSLib [7]

An explicitly parallel version of the `GATHER_DOUBLE` module procedure above now becomes:

```
BUFFER_CELLS: do c = 1, Trace%N_Duplicate
   BUFFER_FACES: do f = 1, SIZE(Src,1)
      Comm_Buffer(f,c) = Src(f,Trace%Duplicate_Indices(c))
   end do BUFFER_FACES
end do BUFFER_CELLS
call PGSLIB_GATHER_BUFFER (Off_Buffer, Comm_Buffer, Trace)
CELL_LOOP: do c = 1, SIZE(Dest,2)
   FACE_LOOP: do f = 1, SIZE(Dest,1)
      if (BTEST(Mesh(c)%Ngbr_PE_Flag, CllNgbr%Bit(f))) then
         Dest(f,c) = Src(Mesh(c)%Ngbr_Face(f), Mesh(c)%Ngbr_Cell(f))
      else
         Dest(f,c) = Off_Buffer(Mesh(c)%Ngbr_Face(f), Mesh(c)%Ngbr_Cell(f))
      end if
   end do FACE_LOOP
end do CELL_LOOP
```

The only difference between this parallel gather operation relative to the previous serial example is that the gather must access a different buffer (`Off_Buffer`) if the requested information is off-processor. Before this operation can be performed, however, the off-processor data must be assimilated and communicated between processors, which is the purpose of the first loop and **PGSLib** call. MPI-based message passing occurs inside the **PGSLib** call.

5 Parallel Linear Solutions with JTpack90 [21]

Our implicit Navier-Stokes and heat transfer/solidification algorithms require the solution of linear systems of equations. A given time step in **Telluride** can require several matrix solutions, so the majority of our solution algorithm is spent in the **JTpack90** linear solver library [21]. This library is written in object-based F90, and is also explicitly parallelized via calls to gather/scatter modules that rely on **PGSLib** [7] to perform the message passing. **Telluride** interfaces to **JTpack90** by linking to its library and "using" its module information files.

We currently solve our systems in parallel over the entire mesh, rather than invoking a Schwarz decomposition [2]. For orthogonal meshes, we store the matrix and use preconditioned CG to solve the system. For generally nonorthogonal, unstructured meshes, we do *not* store the matrix and use preconditioned GMRES to solve the system. In all cases, we interface with **JTpack90** in *matrix-free* form, i.e., matrix-vector multiplication is performed with procedures provided by **Telluride**. All matrix-vector multiplications are therefore performed in **Telluride**, enabling control over indirect addressing (hence their parallelization). This also avoids having to assimilate and store the matrix, which for a general unstructured mesh is often intractable, especially for our current least-squares Laplacian operator [1].

TABLE 1
Chalice solidification parallel efficiencies on a shared memory system.[a]

Processors	CPU Time (μs/cell/cycle)	Efficiency
1	5013	1.00
2	2169	1.15
4	1237	1.03
8	721	0.87

[a]300 MHz Digital AlphaServer 8400

We have found preconditioning the GMRES solution of our least-squares Laplacian operator with a low-order operator (one that assumes the mesh to be orthogonal and simply-connected) to be quite effective. We have additionally found that solving the preconditioner equation with a loosely-converged CG algorithm yields an order of magnitude speedup over more traditional preconditioning alternatives.

6 Numerical Example: Copper Chalice Solidification

We now present evidence for the excellent parallel efficiencies realized in a real-world **Telluride** casting simulation of the cooling and solidification of a copper "chalice". The simulation is performed on a multi-processor shared memory[3] system. The interested reader should also consult reference [21] for additional parallel efficiencies obtained for **Telluride** implicit heat conduction simulations on both shared and distributed memory systems.

The copper chalice was cast at a LANL foundry in support of the inertial confinement fusion program. It is essentially a hemispherical shell (two inch diameter) gated at its pole with a cylindrical "hot top". The hot top serves to continuously supply liquid copper to the hemispherical shell during filling/solidification (to avoid shrinkage defects). The hot top is then cut away and machined after solidification to give the final product (the hemispherical shell).

To date, two single-processor chalice simulations have been performed: (1) isothermal filling of the mold cavity (neglecting heat transfer), and (2) cooling/solidifying of the quiescent liquid copper subsequent to fill. One quadrant of the full geometry is simulated, with the geometric model and computational mesh (6480 unstructured hex elements) being generated with the I-DEAS commercial software package. Space unfortunately does not permit including any chalice simulation results, so the reader is encouraged to consult the **Telluride** home page[4] for graphical results (including animations).

A higher-resolution result (46,386-cell quadrant) is easily achieved with a parallel chalice simulation. Using the Chaco [9] decomposition software, we decompose the mesh into an arbitrary number of submeshes, depending upon the number of processors available to do the problem (see [21] for a mesh decomposition example). As Table 1 indicates, excellent parallel efficiencies are realized for this simulation, which is a good example of the type of parallel casting simulation **Telluride** must perform efficiently (as opposed to an idealized heat conduction problem [21]). Based on these preliminary results, we expect high performance for our parallel unstructured mesh casting simulations provided we make use of intelligent mesh decomposition algorithms [9, 10]. We expect further performance

[3]300 MHz Digital AlphaServer 8400 (see `www.dec.com/info/alphaserver/products.html`)
[4]`http://gnarly.lanl.gov/Telluride/Telluride.html`

improvements to result from additional single-processor optimization and the load balancing of localized models such as interface tracking and phase change.

References

[1] T. J. Barth, *Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations*, Technical Report N92-27677, NASA Ames Research Center, Moffett Field, CA, 1992.

[2] T. J. Barth, *Parallel CFD Algorithms on Unstructured Meshes*, Technical Report AGARD Publication R-807, lecture notes presented at the VKI/NASA/AGARD Lecture Series on Parallel Computing, 1995.

[3] C. Beckermann and C. Y. Wang, *Multiphase/Scale Modeling of Alloy Solidification*, Annual Review of Heat Transfer, 6 (1995), pp. 115–197.

[4] J. B. Bell and D. L. Marcus, *A Second-Order Projection Method for Variable Density Flows*, Journal of Computational Physics, 101 (1992), pp. 334–348.

[5] J. U. Brackbill, D. B. Kothe, and C. Zemach, *A Continuum Method for Modeling Surface Tension*, Journal of Computational Physics, 100 (1992), pp. 335–354.

[6] T. M. R. Ellis, I. R. Philips, and T. M. Lahey, *Fortran 90 Programming*, Addison-Wesley, Reading, MA, 1994.

[7] R. C. Ferrell, D. B. Kothe, and J. A. Turner, *Developing Portable, Parallel Unstructured Mesh Simulations*, in Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing (this conference), Minneapolis, MN, 1997.

[8] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge,, MA, 1994.

[9] B. Hendrickson and R. Leland, *The Chaco User's Guide: Version 2.0*, Technical Report SAND94-2692, Sandia National Laboratories, Albuquerque, NM, 1995.

[10] G. Karypis and V. Kumar, *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995 (see also `www.cs.umn.edu/~karypis/metis/metis.html`).

[11] D. B. Kothe and R. C. Mjolsness, *RIPPLE: A New Model for Incompressible Flows with Free Surfaces*, AIAA Journal, 30 (1992), pp. 2694–2700.

[12] D. B. Kothe, et al., *Computer Simulation of Metal Casting Processes: A New Approach*, Technical Report LALP-95-197, Los Alamos National Laboratory, Los Alamos, NM, 1995.

[13] D. B. Kothe, et al., *Volume Tracking of Interfaces Having Surface Tension in Two and Three Dimensions*, Technical Report AIAA 96-0859, presented at the 34th Aerospace Sciences Meeting and Exhibit, Reno, NV, 1996.

[14] R. Lutowski, *Object-Oriented Software Development with Traditional Languages*, Fortran Forum, 14 (1995), pp. 13–15.

[15] S. McConnell, *Code Complete*, Microsoft Press, Redmond, WA, 1993.

[16] S. McConnell, *Rapid Development*, Microsoft Press, Redmond, WA, 1996.

[17] J. Ni and C. Beckermann, *Modeling of Globulitic Alloy Solidification With Convection*, Journal of Materials Processing and Manufacturing Science, 2 (1993), pp. 217–231.

[18] A. V. Reddy, D. B. Kothe, and C. Beckermann, *High Resolution Finite Volume Simulations of Mold Filling and Binary Alloy Solidification on Unstructured 3-D Meshes*, presented at the 4th Decennial International Conference on Solidification Processing, Univ. of Sheffeld, UK, 1997.

[19] W. J. Rider, D. B. Kothe, S. J. Mosso, J. H. Cerutti, and J. I. Hochstein, *Accurate Solution Algorithms for Incompressible Multiphase Flows*, Technical Report AIAA 95-0699, presented at the 33rd Aerospace Sciences Meeting and Exhibit, Reno, NV, 1995.

[20] J. S. Saltzman, *An Unsplit 3-D Upwind Method for Hyperbolic Conservation Laws*, Journal of Computational Physics, 115 (1994), pp. 153–167.

[21] J. A. Turner, R. C. Ferrell, and D. B. Kothe, *JTpack90: A Parallel, Object-Based, Fortran 90 Linear Algebra Package*, in Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing (this conference), Minneapolis, MN, 1997.

[22] K. D. Wampler, *The Object-Oriented Programming Paradigm (OOPP) and FORTRAN Programs*, Computers in Physics, July/August, (1990), pp. 385–394.